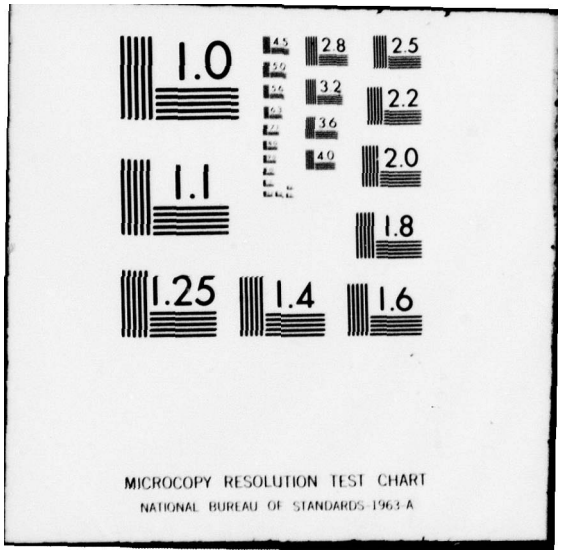
AD-A060 793 STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE F/G 9/2
SOFTWARE RESTYLING IN GRAPHICS AND PROGRAMMING LANGUAGES.(U)
SEP 78 E GROSSE DAHC04-75-G-0185
UNCLASSIFIED STAN-CS-78-663 NL

| OF |
AD
A060793

END
DATE
FILMED
01-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LEVEL II

(12) NW

SOFTWARE RESTYLING IN GRAPHICS AND PROGRAMMING LANGUAGES

by

Eric Grosse

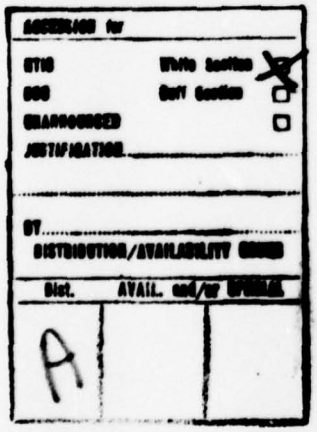
STAN-CS-78-663
SEPTEMBER 1978

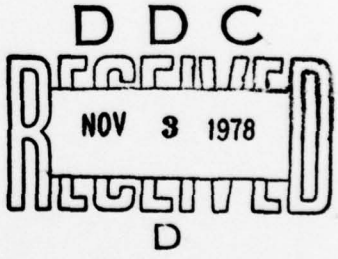
COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

SOFTWARE RESTYLING IN GRAPHICS AND PROGRAMMING LANGUAGES

by

Eric Grosse*





*Department of Computer Science, Stanford University, Stanford, CA 94305

This paper was presented at the 1978 Army Numerical Analysis and Computers Conference in Huntsville, Alabama, March 1, 1978.

Research supported in part under Army Research Grant DAHC04-75-G-0185 - Rochester NY. and in part under National Science Foundation Grant MCS75-13497-A01.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-78-663	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER rept.
4. TITLE (and Subtitle) SOFTWARE RESTYLING IN GRAPHICS AND PROGRAMMING LANGUAGES.		5. TYPE OF REPORT & PERIOD COVERED Technical, September 1978
		6. PERFORMING ORG. REPORT NUMBER STAN-CS-78-663
7. AUTHOR(s) Eric Grosse		8. CONTRACT OR GRANT NUMBER(s) DAHC04-75-G-0195
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department Stanford University Stanford, California 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Research Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE September 1978 13. NUMBER OF PAGES 30
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) 33p.		15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Releasable without limitations on dissemination.

DISTRIBUTION STATEMENT A
Approved for public release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The value of large software products can be cheaply increased by adding restyled interfaces that attract new users. As examples of this approach, a set of graphics primitives and a language precompiler for scientific computation are described. These two systems include a general user-defined coordinate system instead of numerous system settings, indention to specify block structure, a modified indexing convention for array parameters, a syntax for n-and-a-half-times'-round loops, and engineering format for real constants; most of all, they strive to be as small as possible.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

094 120

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SOFTWARE RESTYLING IN GRAPHICS AND PROGRAMMING LANGUAGES

Eric Grosse
Computer Science Department
Stanford University
Stanford CA 94305

ABSTRACT. The value of large software products can be cheaply increased by adding restyled interfaces that attract new users. As examples of this approach, a set of graphics primitives and a language precompiler for scientific computation are described. These two systems include a general user-defined coordinate system instead of numerous system settings, indention to specify block structure, a modified indexing convention for array parameters, a syntax for n-and-a-half-times-'round loops, and engineering format for real constants; most of all, they strive to be as small as possible.

0.0 PHILOSOPHY. Kernighan and Plauger [1976] describe explicitly and by example three precepts of the Software Tools philosophy:

- trim out the inessentials
- build it adaptively
- let someone else do the hard part

Two more examples, driven by the same philosophy, are given below. The basic idea is to obtain high leverage by taking an existing, powerful piece of software and make it useful to more people by designing a new interface. Webster's calls this process facelifting: "a restyling intended to increase comfort or salability."

1.0 JUSTIFICATION FOR STILL ANOTHER PROGRAMMING LANGUAGE. Fortran will no doubt remain for many years the most important programming language for scientific computation. When used carefully and with discipline, it yields remarkably portable codes; this is its greatest virtue. But, as programmers have complained for years, it also has many faults:

- awkward syntax for statements, strings, names
- primitive control structures
- DO loop restrictions
- no macros

Fortran preprocessors, such as MORTRAN [Cook+Shustek 1975], have eliminated many of these disadvantages and therefore have become very popular. Unfortunately, they reduce portability somewhat, since either the preprocessor must be installed at the new site

or illegible 'object' Fortran sent there. More importantly, such preprocessors have only a minor effect on inherent problems of Fortran:

- dynamic allocation is either unavailable or requires the use of rather confusing tricks
- no PROCEDURE VARIABLE type
- no STRUCTURE type
  (labelled common blocks, since they do not use the combinatorial possibilities of procedure parameterization, are less flexible.)
- no 0-origin indexing
- array bound information is not automatically passed
- no vector operations
- no recursion

The PORT library makes dynamic allocation one of its most advertised features: "We have found that use of dynamic storage allocation in PORT leads to more clearly structured programs, cleaner calling sequences, improved memory utilization, and better error detection." [Fox+Hall+Schryer 1977] Adding a stack to Fortran is a messy affair, however, as shown in figure 1, which contains two alternate methods in PORT for allocating an

```
SUBROUTINE LBB(A,N)

COMMON /CSTAK/DSTAK(500)

DOUBLE PRECISION DSTAK
INTEGER ISTAK(1000)
REAL A(1)
REAL RSTAK(1000)

EQUIVALENCE (DSTAK(1),ISTAK(1))
EQUIVALENCE (DSTAK(1),RSTAK(1))

II = ISTKGT(2*N,2)
IR = ISTKGT(N,3)

.
.

{ code referring to RSTAK(IR+n) and ISTAK(II+m)
  probably ending with code to store the stuff
  from the real scratch storage into array A  }

.
.

CALL ISTKRL(2)

RETURN
END
```

```
SUBROUTINE LBB(A,N)

COMMON /CSTAK/DSTAK(500)

DOUBLE PRECISION DSTAK
INTEGER ISTACK(1000)
REAL A(1)
REAL RSTAK(1000)

EQUIVALENCE (DSTAK(1),ISTAK(1))
EQUIVALENCE (DSTAK(1),RSTAK(1))

II = ISTKGT(2*N,2)
IR = ISTKGT(N,3)

CALL L1BB(A,ISTAK(II),RSTAK(IR),N)

CALL ISTKRL(2)
RETURN
END
```

figure 1

INTEGER and REAL array.

Other proposals are even more complicated. (After a 7 page description of DYNOSOR, Huybrechts[1977] states: "This paper gives only the basic features of the DYNOSOR system. A more sophisticated use allows the user, once he is familiarized with the system, to improve greatly the speed of programs using it.")

PL/I, which is now becoming fairly widely available in some form, overcomes all these difficulties. However, so huge a language tends to overwhelm people, and because of tricky precision rules, silent type conversions (as in I=J=0;), and the like, learning only part of the language is dangerous.

Other languages, while beautifully designed, have their own flaws. For example, Algol W does not have a robust interface to Fortran; in addition to this [Wohilner 1977], Pascal places painful restrictions on arrays.

1.1 T. Thus another approach seems warranted, which can combine the needed features of PL/I, the deliberate syntax of ALGOL, and the low implementation cost of the Fortran preprocessors. Such an approach has produced the language T, intended to assist in the implementation and documentation of algorithms for scientific computation. The principal aims have been ease of reading and writing, low implementation cost, and reasonable efficiency.

Appendix T gives the formal language proposal, specifying the syntax according to Wirth's proposal [1977]. Since T is similar to Fortran, Algol 60, and PL/I, a complete specification of the semantics may be omitted without confusion. To provide the heuristics behind the design choices and to give an overview of the language, various aspects of the folowing example will be discussed.

```
TRIPEAK
   # example of T and G systems;
   # various views of the sum of three Gaussian peaks;
   #   Eric Grosse   Stanford University

   REAL: AZIM, ELEV,      # VIEWING ANGLES FOR SURFACE PLOT
         RELERR, ABSERR,  # ERROR TOLERANCES FOR ODE
         T, TOUT,         # INDEPENDENT VARIABLES OF TRAJECTORY
         NORMYP           # 2 NORM OF THE GRADIENT
   REAL(2): LL, UR,       # CORNERS OF RECTANGULAR DOMAIN OF FUNCTION
            ORIGIN,       # FOCAL POINT FOR SURFACE PLOT
            XO, SCALE,    # COORDINATE TRANSFORMATION PARAMETERS
            Y, YP         # LOCATION AND GRADIENT FOR TRAJECTORY
   REAL(142): ODEWORK
   INTEGER(5): ODEIWORK
```

```
DEFINE(P,20)            # density of F samples;
REAL(-P:P,-P:P): F TABLE
REAL(3): LEVEL          # CONTOUR LEVELS
INTEGER: I, J,
         IFLAG          # DIAGNOSTICS FLAG FOR ODE
STRUCTURE: PARAM        # LOCATIONS, HEIGHTS, AND WIDTHS OF PEAKS
   REAL(3,2): X
   REAL(3): H, W
STRUCTURE: PF           # PLOT FILE
   INTEGER(500): WORK
PROCEDURE: GOPEN, GCLOSE, GPICT, GCONT, GSURF, GLTYPE,
     GJUMP, GDRAW, GTRAN1
FORTRAN PROCEDURE: ODE, DF, STASH
PROCEDURE () REAL: F

# SET UP PARAMETERS
BLANK SEPARATION (2)
REAL DIGITS(3)
GET DATA(AZIM,ELEV)
PUT DATA(AZIM,ELEV)
X(1,1) := 0
X(1,2) := 0.5
X(2,1) := -0.43301 2702
X(2,2) := -0.25
X(3,1) := -X(2,1)
X(3,2) := X(2,2)
PUT DATA ARRAY(X)
GET ARRAY(H)
PUT DATA ARRAY(H)
GET ARRAY(W)
PUT DATA ARRAY(W)
STASH(X,H,W)
FOR( -P <= I <= P )
   Y(1) := FLOAT(I) / P
   FOR( -P <= J <= P )
      Y(2) := FLOAT(J) / P
      F TABLE(I,J) := F(Y,PARAM)

# SURFACE PLOT
GOPEN('VEP12PF',PF)
GPICT(PF)
LL := -1
UR := 1
ORIGIN := 0.5
GSURF(LL,UR,FTABLE,AZIM,ELEV,ORIGIN,0.25,PF)
```

```
    # CONTOUR PLOT
    GPICT(PF)
    SCALE := 0.3333
    X0 := -0.5/SCALE(1)
    GTRAN1(X0,SCALE,PF)
    GET ARRAY(LEVEL)
    PUT DATA ARRAY(LEVEL)
    GCONT(LL,UR,FTABLE,LEVEL,PF)
    GLTYPE('DOT',PF)
    GET ARRAY(LEVEL)
    PUT DATA ARRAY(LEVEL)
    GCONT(LL,UR,FTABLE,LEVEL,PF)

    # COMPUTE AND PLOT TRAJECTORY
    RELERR := 10(-6)
    GLTYPE('SOLID',PF)
    ABSERR := 10(-6)
    WHILE( ¬ END OF INPUT )
       GET ARRAY( Y )
       PUT DATA ARRAY( Y )
       T := 0
       GJUMP(Y,PF)
       IFLAG := 1
       WHILE( NORMYP > 1(-3) & 1<=IFLAG & IFLAG<=3 )
          TOUT := T + 10(-3)/NORMYP
          CDE(DF,2,Y,T,TOUT,RELERR,ABSERR,IFLAG,ODEWORK,ODEIWORK)
          CASE
             2 = IFLAG
                GDRAW(Y,PF)
             3 = IFLAG
                PUT('ODE DECIDED ERROR TOLERANCES WERE TOO SMALL.')
                PUT('NEW VALUES:')
                PUT DATA(RELERR,ABSERR)
             ELSE
                PUT('ODE RETURNED THE ERROR FLAG:')
                PUT DATA(IFLAG)
          FIRST
          DF(T,Y,YP)
          NORMYP := NORM2(YP)
    GCLOSE(PF)

F ( Y, PARAM ) Z
    REAL(): Y
    REAL: Z, NORMSQ
    STRUCTURE: PARAM
       REAL(3,2): X
       REAL(3): H, W
    INTEGER: I
    Z := 0
    FOR( 1 <= I <= 3 )
       NORMSQ := (Y(1)-X(I,1))**2 + (Y(2)-X(I,2))**2
       Z := Z + H(I)*EXP(-0.5*W(I)*NORMSQ)
```

1.2 CONTROL AND OTHER SYNTAX. Perhaps the most striking feature the Algol veteran sees in this example is the complete absence of BEGINs and ENDs. Not only is the text indented, but the indention actually specifies the block structure of the program. Such a scheme was apparently first proposed by Landin [1966]. Except for an endorsement by Knuth [1974], the idea seems to have been largely ignored.

Ideally, the text editor would recognize tree-structured programs [Hansen 1971]. In practice, text editors tend to be line oriented so that moving lines about in an indented program requires cumbersome manipulation of leading blanks. Therefore the current implementation of T uses BEGIN and END lines, translating to indention on output. Thus the input

```
STRUCTURE: PARAM
((
REAL(3,2): X
REAL(3): H, W
))
```

produces the output

```
STRUCTURE: PARAM
   REAL(3,2): X
   REAL(3): H, W
```

Whatever the implementation, the key idea is to force the block structure and the indention to be automatically the same, and to reduce clutter from redundant keywords.

Blanks are insignificant outside of strings. Mathematical tables have long used blanks inside numeric constants, as in

```
PI := 3.14159 26535 89793
```

for readability. Blanks in identifiers also can improve readability, while reducing the chance of misspelling and easing the pain of name length restrictions imposed by the local operating system.

In accordance with the recommendations of Scowen+Wichmann [1973], comments start with a special character, #, and run to the end of the physical line.

The small reserved word list eliminates the need for a stropping convention. The psychological advantages of this approach have been elaborated by Hansen [1973].

The form of the assignment and procedure call statements follows the clean, clear style of Algol 6C. To make macros more understandable, their syntax and semantics match those of procedures as closely as possible.

In addition to normal statement sequencing and procedure calls, three control structures are provided. The CASE and WHILE statements are illustrated in this typical program segment:

```
WHILE( NORMYP > 1(-3) & 1<=IFLAG & IFLAG<=3 )
   TOUT := T + 10(-3)/NORMYP
   ODE(DF,2,Y,T,TOUT,RELERR,ABSERR,IFLAG,ODEWORK,ODEIWORK)
   CASE
      2 = IFLAG
         GDRAW(Y,PP)
      3 = IFLAG
         PUT('ODE DECIDED ERROR TOLERANCES WERE TOO SMALL.')
         PUT('NEW VALUES:')
         PUT DATA(RELERR,ABSERR)
      ELSE
         PUT('ODE RETURNED THE ERROR FLAG:')
         PUT DATA(IFLAG)
   FIRST
   DF(T,Y,YP)
   NORMYP := NORM2(YP)
```

The CASE statement is modelled after the conditional expression of LISP; the boolean expressions are evaluated in sequence until one evaluates to YES, or until ELSE is encountered. The use of indention makes it easy to visually find the relevant boolean expression and the end of the statement.

One unusual feature of the WHILE loops is the optional FIRST marker, which specifies where the loop is to be entered. In the example above, the norm of the gradient, NORMYP, is computed before the loop test is evaluated. Thus the loop condition, which often provides a valuable hint about the loop invariant, appears prominently at the top of the loop, and yet the common n-and-a-half-times-'round loop can still be easily expressed.

The FOR statement adheres as closely as practical to common mathematical practice.

```
FOR( 1 <= I <= 3 )
   NORMSQ := (Y(1)-X(I,1))**2 + (Y(2)-X(I,2))**2
   Z := Z + H(I)*EXP(-0.5*W(I)*NORMSQ)
```

Several years experience with these control constructs has demonstrated them to be adequately efficient and much easier to maintain than the alternatives.

Procedure nesting is not used for two reasons. First, textual nesting that extends over many pages is difficult for a human to keep track of. Second, programs typically contain several high level procedures calling a single primitive, so a tree representation is inappropriate anyway.

By removing the nesting of procedures, however, we worsen the problem of entry point hiding that arises when combining programs from many sources into a single library. A solution to this problem is to have an official name for each procedure, coded along the lines of IMSL, and also a more mnemonic nick name (which users can pick for themselves if they like). The macro

processor which is built into T can then be used to change all occurences of the nick names into the corresponding official names.

1.3 DECLARATIONS. The fundamental scalar types are INTEGER, REAL, and COMPLEX, from which arrays and structures may be built up. As the example

```
REAL(-P:P,-P:P)
```

illustrates, general upper and lower bounds are allowed.

The upper bound expression is omitted for a formal array parameter, so that an appropriate value can be taken from the length of the corresponding actual array argument. The origin of an actual array argument need not match the origin of the corresponding formal array parameter. For example, if the actual argument A was declared REAL(0:7): A and the formal parameter B was declared REAL(): B, then B(8) will correspond to A(7). Most languages, when they allow lower bounds at all, do not permit this flexibility, which is used in the example program when a matrix with lower bound -P is passed to a general purpose library routine which assumes a lower bound of 0.

Structures of arbitrary depth may be declared. As the examples

```
STRUCTURE: PARAM
   REAL(3,2): X
   REAL(3): H, W
STRUCTURE: PP
   INTEGER(500): WORK
```

suggest, structures are useful passing collections of related data, without the need for long parameter lists. This makes feasible the prohibition of global variables in a drastic attempt to narrow and make more explicit the interface between procedures. Euclid [Popek+others 1977] has emphasized the importance of visibility of names.

The graphics procedures which use the WORK vector of the example are able to divide up the space into convenient units. This capability, which would be possible in PL/I only through the use of pointers, encourages information hiding and abstraction.

PROCEDURE VARIABLEs allow the names of procedures to be saved, an essential feature for applications like the user-specified coordinate transformation described in the graphics system below.

The importance of existing Fortran software is recognized by providing for FORTRAN PROCEDUREs as an integral part of the language. The current implementation of T performs this linkage in a more efficient way than the naive user of PL/I would be likely to discover.

A novel syntax is introduced for function returns. Since procedures may be recursive, Fortran's convention of using the function name as variable cannot be followed. Instead, the procedure header declares a return variable just like any other parameter:

```
F ( Y, PARAM ) Z
   REAL(): Y
   REAL: Z
   ...
```

1.4 INPUT/OUTPUT. Beginners often find Fortran's input/output the most difficult part of the language, and even seasoned programmers are tempted to just print unlabelled numbers, often to more digits than justified by the problem, because formatting is so tedious. PL/I's list and data directed I/O is so much easier to use that it was wholeheartedly adopted in T. By providing procedures for modifying the number of decimal places and the number of separating blanks to be output, no edit-directed I/O is needed. Special statements are provided for array I/O so that, unlike PL/I, arrays can be printed in orderly fashion without explicit formatting.

Since almost as much time is spent in scientific computation staring at pages of numbers as at pages of program text, much thought was given to the best format for displaying numbers.

In accordance with the "engineering format" used on Hewlett-Packard calculators and with standard metric practice [GM Service Section 1977], exponents are forced to be multiples of 3. As figure 2, an excerpt from the example program's output, shows, this convention has a histogramming effect that concentrates the information in the leading digit, as opposed to splitting it between the leading digit and the exponent, which are often separated by 14 columns. The use of parentheses to surround the exponent, like the legality of imbedded blanks, was suggested by mathematical tables. This notation separates the exponent from the mantissa more distinctly than the usual E format.

1.5 DISCUSSION.

Following Kernighan+Plauger [1976], the initial implementation is unsophisticated [Comer 1978]. Nevertheless, the preprocessing is less costly than the PL/I compile, so the overall results are quite satisfactory. (The evaluation looks even better if one compares PL/I + T against PL/I + PL/I's macro preprocessor.) Most of the processor cost lies in basic I/O; by integrating the macro processor with the language translator, this cost has been minimized. [Kantorowitz 1976] Much of the two-man-months spent in implementation were spent in understanding nooks and crannies of PL/I.

```
    53.5106 (-03)          5.35106E-02
    51.3109 (-03)          5.13109E-02
    46.7211 (-03)          4.67211E-02
    40.6514 (-03)          4.06514E-02
    33.7636 (-03)          3.37636E-02
    26.4908 (-03)          2.64908E-02
    18.9806 (-03)          1.89808E-02
    11.3461 (-03)          1.13461E-02
     3.63508(-03)          3.63508E-03
-    4.12944(-03)         -4.12944E-03
-   11.9123 (-03)         -1.19123E-02
-   19.7092 (-03)         -1.97092E-02
-   27.5248 (-03)         -2.75248E-02
-   35.3243 (-03)         -3.53243E-02
-   43.1176 (-03)         -4.31176E-02
-   50.9066 (-03)         -5.09066E-02
-   58.6841 (-03)         -5.86841E-02
-   66.4483 (-03)         -6.64483E-02
-   74.1973 (-03)         -7.41973E-02
-   81.9297 (-03)         -8.19297E-02
-   89.6443 (-03)         -8.96443E-02
-   97.3401 (-03)         -9.73401E-02
-105.016  (-03)           -1.05016E-01
-112.670  (-03)           -1.12670E-01
-120.302  (-03)           -1.20302E-01
-127.910  (-03)           -1.27910E-01
-135.493  (-03)           -1.35493E-01
-143.050  (-03)           -1.43050E-01
```

figure 2

T is not intended to replace any existing languages. For distributing mathematical software, Fortran remains the only practical medium; for character processing, something like PL/I or SNOBOL should be used. Still, for the bulk of scientific computation, T ought to be the easiest to use, particularly since it coexists comfortably with Fortran and PL/I. On the other hand, one can imagine ways that T might be improved, as well. Features omitted for ease of implementation include:

- trimmed arrays, like X(2:N)
- procedure results of general type
- conditional boolean operators that do not evaluate their arguments when it is possible to avoid doing so
- a swap operator

For other features, no entirely satisfying design was apparent:

- strings
- more general procedure calls (such as indefinite number and type of arguments)
- a means of constructing arrays directly from components, as

a string constant constructs a string from individual characters
- a means of specifying the invocation graph of who calls whom

Perhaps the most fundamental though unavoidable flaw is that, unlike LISP, the language is not trivial, and therefore programs cannot be trivially manipulated.

2.0 JUSTIFICATION FOR STILL ANOTHER SET OF GRAPHICS PRIMITIVES.

The next example of restyling is a simple but reasonably complete interface for noninteractive device-independent graphics. In addition to the basic line drawing primitives, higher level procedures are provided for displaying functions of one or two variables. This interface has been implemented as a library of PL/I procedures which call the SLAC Unified Graphics package written by Robert Beach [1978].

Unified Graphics, with its emphasis on the ability to drive displays like the IBM 2250, is troublesome to use directly for function plots and the like. In contrast, Top Drawer, another graphics system at SLAC, allows for function plots but little else. The collection described in detail in Appendix G is meant to strike a useful balance between these two extremes, and contains most of the features of DISSPLA important for scientific computation.

2.1 ESTABLISHING THE ENVIRONMENT.

The following excerpt from the example program given in section 1.1 above illustrates typical preparation for plotting:

```
STRUCTURE: PF          # PLOT FILE
   INTEGER(500): WORK
REAL(2): LL, UR,       # CORNERS OF RECTANGULAR DOMAIN
         ORIGIN,       # FOCAL POINT FOR SURFACE PLOT
         X0, SCALE     # COORDINATE TRANSFORMATION PARAMETERS
GOPEN('VERSAFF',PF)
GPICT(PF)
SCALE := 0.3333
X0 := -0.5/SCALE(1)
GTRAN1(X0,SCALE,PF)
```

The plot area PF is used to remember various options and to buffer low level plotter instructions. This work area is initialized by the GOPEN call, which specifies the output device. (In the current implementation, no corresponding JCL changes are necessary.) The ease with which devices may be changed is very useful in tuning a plot for publication.

For compatibility with numerical procedures, REAL variables are in full precision, not short. At the start of each new picture, which might be a screenful on a CRT or an 8.5 by 11" page on an electrostatic plotter, GPICT is called.

All plotting is done relative to a user coordinate system, which is specified by calling

```
GTRAN( F, PP )
```

where F is the name of a procedure which, when called in the form

```
F( X, W, PP )
  with
    REAL(N): X        N<=10
    REAL(2): W
```

will map the point X in user coordinates into a point W in the unit square [0,1]x[0,1]. Normally W(1) is thought of as horizontal and W(2) as vertical. By extending PP, the user can pass parameters to F. For convenience, the default transformation maps

```
W := SCALE * ( X - X0 )
```

2.2 DRAWING, DIMENSIONING, AND FUNCTION GRAPHING. The basic drawing commands are GJUMP, GDRAW, and GTEXT for drawing lines and adding text. If a nonlinear coordinate system has been specified, GDRAW produces a piecewise linear approximation to the implied curve.

A procedure GGRAPH is provided which automatically samples function values, sets up an appropriate scaling, graphs the function, and dimensions the graph using round numbers in a style consistent with the format used by T. Figure 3, taken from Chan [1978], is a typical plot.

The scheme for choosing round numbers is based on the algorithm by Dixon+Kronmal [1965]. Experience and an informal survey of what people would accept as being "round numbers" led to various refinements. As in Unified Graphics, the choice is optimized over a reasonable number of major tick marks. The total number of tic marks, major and minor, is not allowed to be either too dense or too sparse. For a while, the number of minor tick marks was chosen so that each interval had length 10**k, but for input data limits (20,70) the resulting tick marks were at (-100,0,100,200), so this rule had to be relaxed to "either length 10**k or midpoint of major interval." If the difference between the data limits is small compared to the magnitude of the limits themselves (as occurs for example in plotting a nearly constant function), then the labels may become unreasonably large. Special provision is made for this case.

Other routines are available for scatter, surface, and contour plots. The contour computation uses piecewise quadratic surface fitting to ensure smooth contours and proper representation of critical points [Marlow+Powell 1976]. Figure 4 presents output from the example program, which computes hill-climbing trajectories for a three-gaussian-peak terrain.

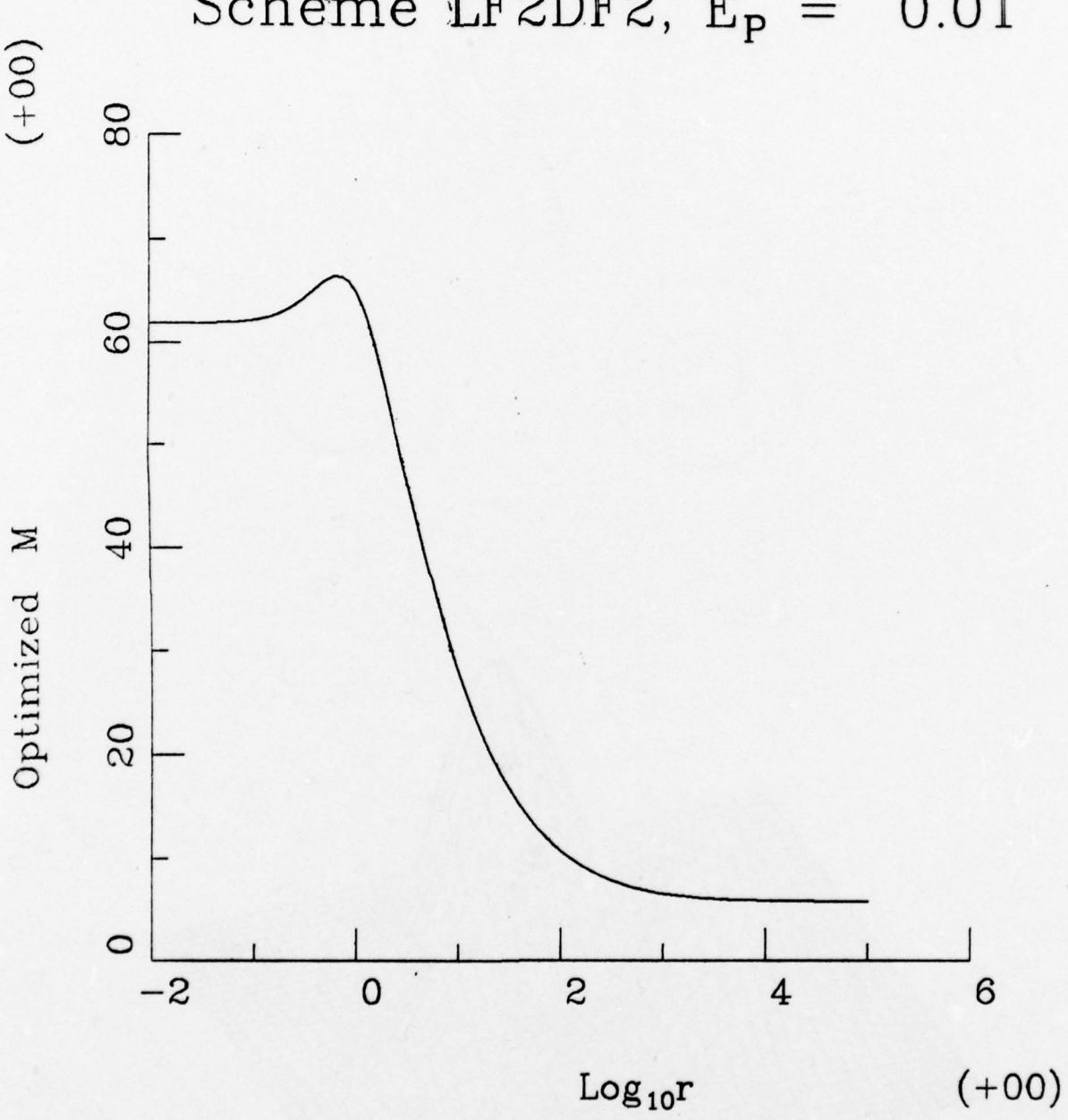

Scheme LF2DF2, E_p = 0.01
(+00)
80
60
40
20
0
Optimized M
−2
0
2
4
6
$Log_{10}r$
(+00)

figure 3

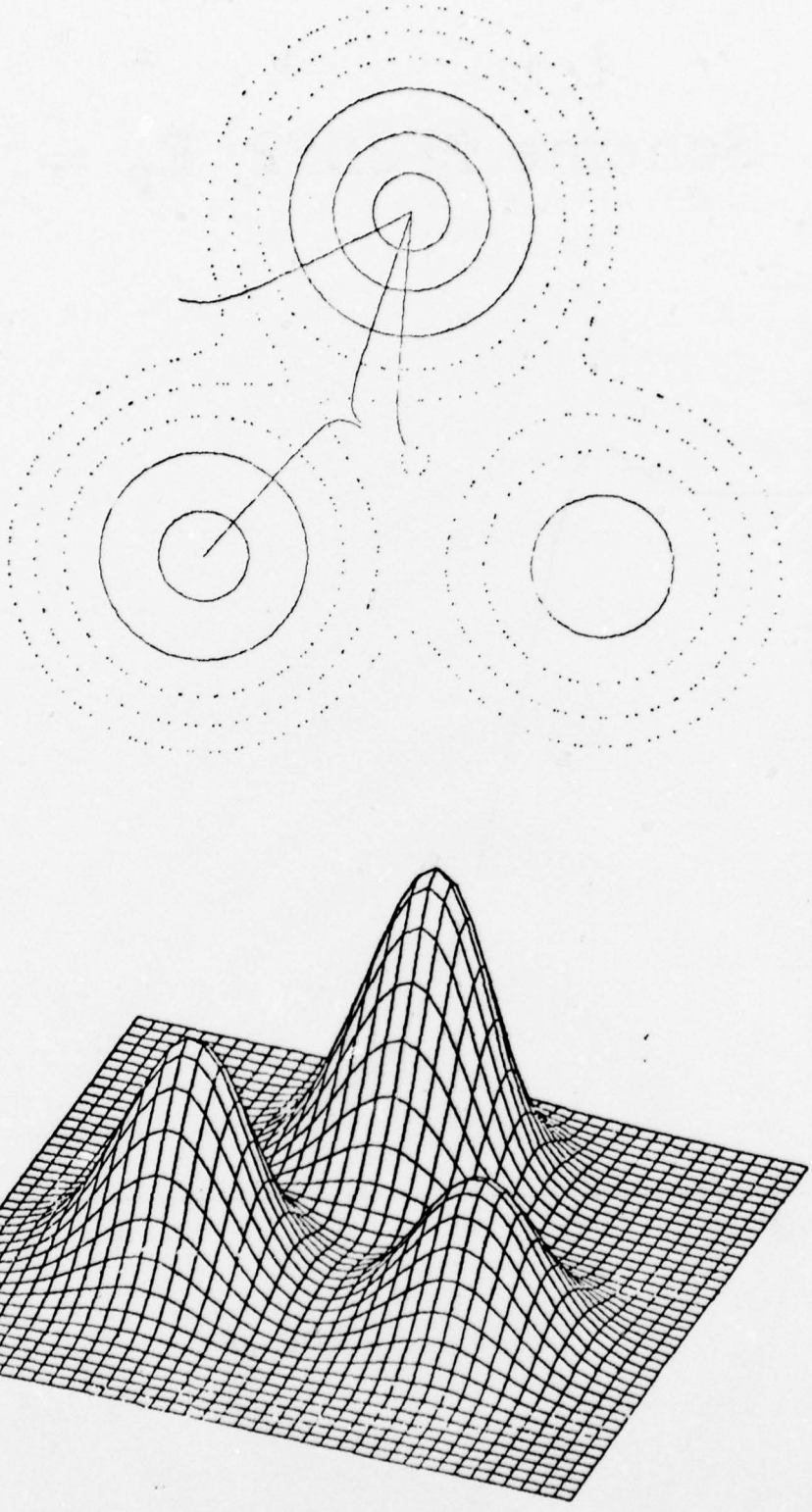


figure 4

CONCLUSION. With a level of effort comparable to writing a Fortran preprocessor, we have created, by compiling into PL/I, a language substantially better than Fortran or its derivatives. Since PL/I problems cannot be altogether avoided by this approach, further work on a language like T could be useful. Perhaps the effort would be better spent on making LISP a practical language for scientific compuation by building on the research in symbolic computation.

Like PL/I, Unified Graphics is good for a wide range of applications. But in practice, many people won't use either. For languages, they stick to Fortran; for graphics, they plot by hand or not at all. In both cases it has proven possible to cheaply restyle the existing system, via a preprocessing phase or driver routines, in order to create more agreeable tools.

ACKNOWLEDGEMENTS. Special thanks go to Bill Coughran for discussions of this report and help with T's realization in a PL/I precompiler. Helpful comments were made by Petter Bjorstad, Dan Boley, Tony Chan, Hector Garcia, Mike Heath, Randy Leveque, and Bob Melville. Support was provided by a National Science Foundation graduate fellowship and grant DRXRO-MA-13292-M from the US Army Research Office; computing was provided at the Stanford Linear Accelerator Center by the Department of Energy.

BIBLIOGRAPHY.

Beach, Robert [Jun 1978]
The SLAC Unified Graphics System: programming manual
Stanford Linear Accelerator Center CGTM 170

Chan, Tony F C [Apr 1978]
Comparison of numerical methods for initial value problems
Stanford Univ PhD thesis

Comer, Douglas [1978]
MOUSE4: an improved implementation of the RATFOR preprocessor
Soft Pract + Exper 8, 35-40

Cook, A James + L J Schustek [Jun 1975]
A user's guide to MORTRAN2
Stanford Linear Accelerator Center CGTM 165

Dixon, W J + F A Kronmal [Apr 1965]
The choice of origin and scale for graphs
J ACM 12, 259-261

Fox, P A + A D Hall + N L Schryer [May 1977]

The PORT mathematical subroutine library
Bell Laboratories Murray Hill Comp Sci Tech Rep 47

GM Service Section [1977]
Metric usage guide
Detroit MI

Hansen, Wilfred J [Jul 1971]
Creation of hierarchic text with a computer display
Argonne National Laboratory ANL 7818

Hansen, Wilfred J [Nov 1973]
A revised Algol 68 hardware representation for ISO-CODE and EBCDIC
Univ Illinois Urbana UIUCDCS R 73 607

Huybrechts, M [Apr 1977]
DYNOSOR a set of subroutines for dynamic memory organization
SIGPLAN Notices Apr 1977, 67-74

Kantorowitz, E [Feb 1976]
Macro processors for efficient program production
European Research Office, US Army

Kernighan, Brian W + P J Plauger [1976]
Software Tools
Addison Wesley

Knuth, Donald E [Dec 1974]
Structured programming with go to statements
Computing Surveys 6, 261-301

Landin, P J [Mar 1966]
The next 700 programming languages
Comm ACM 9, 157-166

Marlow, S + M J D Powell [Apr 1976]
A Fortran subroutine for plotting the part of a conic that is inside a given triangle
UK AERE Harwell R8336

Mohilner, Patricia R [1977]
Using PASCAL in a FORTRAN environment
Soft Pract + Exper 7, 357-362

Popek, G J + J J Horning + B W Lampson + J G Mitchell + R L London [Mar 1977]
Notes on the design of EUCLID
SIGPLAN Notices Mar 1977, 11-18

Scowen, R S + B A Wichmann [May 1974]
The definition of comments in programming languages
Soft Pract + Exper 4, 181-188

Wirth, Niklaus [Nov 1977]
What can we do about the unnecessary diversity of notation for syntactic definitions?
Comm ACM 20,822-823

APPENDIX T

Report on the programming language T

"trim: free from anything extraneous;
having clean lines or proper proportion;
the state of readiness for action or use;"
Webster's Third New International

"Everything should be as simple as possible,
but no simpler."
Einstein

"What it lies in our power to do,
it lies in our power not to do."
Aristotle

"In all spheres, the true craftsman is
the one who thoroughly understands his tools."
Hoare

TOKENS. Program text is made up of the following tokens:

keyword - one of the following:

CASE	NO
COMPLEX	PROCEDURE
ELSE	PROCEDURE VARIABLE
FIRST	PUT
FOR	PUT ARRAY
FORTRAN PROCEDURE	PUT DATA
FORTRAN PROCEDURE VARIABLE	PUT DATA ARRAY
GET	REAL
GET ARRAY	STRUCTURE
GET DATA	WHILE
INTEGER	YES

identifier - a letter optionally followed by more letters and digits.

integer-constant - one or more digits

real-constant - one or more digits , a ".", possibly more digits, a "(", possibly a "-", one or more digits, and a ")". Either the decimal point and succeeding digits or the parentheses and exponent, but not both, may be omitted. Thus 1., 0.23, 6.22(-23), 1(-6), and 3.14159(+00) are all legal real-constants.

string-constant - a sequence of characters enclosed in apostrophes. Apostrophes are not allowed in the string proper.

delimiter - one of the following:
eor ( ) $ @ : . , & | ¬ + - * ** / := = ¬= < <= > >=
(The last six are called relational-operators.)

Except in string-constants, blanks are insignificant and may be used freely for clarity. Text from a "#" up to the next "eor" (end-of-record) is treated as a comment.

PROCEDURES.

```
program   =  {procedure} .
procedure =  identifier [ "(" identifiers ")" [result] ] "eor"
             begin
             {declaration}
             {statement}
             end .
identifiers =  {identifier,} identifier .
result    =  identifier .
begin     =  "((" "eor" .
end       =  "))" "eor" .
```

Parameters are passed using call-by-reference. "result" may be used just like any other formal parameter, in particular as a destination, but must be a scalar.

If the parameter list is omitted, the procedure is assumed to be the top level main program.

DECLARATIONS.

```
declaration  =  scalar-type [ "(" bounds ")" ] ":" identifiers "eor"
             |  "STRUCTURE"  ":" identifiers "eor"
                    begin
                    {declaration}
                    end
             |  ["FORTRAN"] "PROCEDURE" ["VARIABLE"]
                    ["()" scalar-type ] ":" identifiers "eor"
scalar-type  =  "INTEGER"
             |  "REAL"
             |  "COMPLEX" .
bounds  =  [bounds,] [expression:] [expression] .
```

No global variables are allowed; communication occurs only through parameter lists. Declarations reserve storage on a stack; the variables are undefined until first assigned to.

If the <expression>: part of a bound is omitted, 1: is assumed. The second <expression> should be omitted for a formal array parameter, and an appropriate value will be taken from the length of the corresponding actual array argument. The origin of an

actual array argument need not match the origin of the corresponding formal array parameter. For example, if the actual argument A was declared REAL(0:7): A and the formal parameter B was declared REAL(): B, then B(8) will correspond to A(7).

STRUCTURES are data areas assumed to be decomposed as indicated in the subdeclarations. Thus if actual parameter A is declared

```
      STRUCTURE: A
         INTEGER: I, K
         REAL: X
         COMPLEX: Z
```

and corresponding formal parameter F is declared

```
      STRUCTURE: F
         INTEGER: J, L
         COMPLEX: W
```

then A.I and F.J correspond, but A.Z and F.W do not. (If F.W is assigned to, both A.X and A.Z may be destroyed.)

A PROCEDURE VARIABLE is a variable that may refer to various actual procedures; in contrast, a PROCEDURE is literally the name of a procedure.

STATEMENTS.

```
statement  =  procedure-call "eor"
  |  destination ":=" expression  "eor"
  |  "GET(" identifiers ")"
  |  "GET ARRAY (" identifer ")"
  |  "GET DATA (" identifiers ")"
  |  "PUT(" arguments ")"
  |  "PUT ARRAY (" destination ")"
  |  "PUT DATA (" arguments ")"
  |  "PUT DATA ARRAY (" destination ")"
  |  "CASE" "eor"
        begin
        {boolean-expression "eor"
          begin
          {statement}
          end }
        ["ELSE" "eor"
          begin
          {statement}
          end  ]
        end
  |  "WHILE(" boolean-expression ")" "eor"
        begin
        {statement}
        ["FIRST" "eor"
        {statement} ]
        end
  |  "FOR("
   (( expression ("<"|"<=") destination ("<"|"<=") expression)
   |( expression (">"|">=") destination (">"|">=") expression) ) "eor"
```

```
begin
{statement}
end    .
```

GET reads from the next record of the input data a sequence of constants, separated by commas. For GET DATA, each value should be prefixed with "identifier :="; the input values need not appear in the same order as the corresponding identifiers in the GET DATA, and if a value is omitted, the variable is left unchanged. PUT DATA and PUT write out the current values of the identifiers, labelled or unlabelled, in an intelligent fashion.

In other languages,

```
CASE
   cond1
      case1
   cond2
      case2
   ELSE
      case3
```

might be written as:

```
IF( cond1 ) THEN
   case1
ELSE IF( cond2 ) THEN
   case2
ELSE
   case3
```

Similarly,

```
WHILE cond
   part a
   FIRST
   part b
```

might be translated as:

```
   GOTO FIRST
TOP:  part a
FIRST: part b
   IF( cond ) THEN GOTO TOP
```

If the FIRST line is omitted, it is assumed to be at the end of the loop; that is, part b is empty.

Finally,

```
FOR( LOW <= I <= HIGH )
   loop
```

would be translated as

```
FOR I = LOW TO HIGH
   loop
```

and

```
FOR( HIGH > I >= LOW )
   loop
```

as

```
FOR I = HIGH-1 BY -1 TO LOW
   loop
```

The destination and expressions controlling a FOR loop may not be modified inside the loop.

```
EXPRESSIONS.
destination  =  {identifier "."} identifier
             |  identifier "(" subscripts ")"
             |  @ procedure-identifier  .
subscripts  =  [subscripts,] [expression]
procedure-call  =  procedure-identifier [ "(" arguments ")" ]  .
procedure-identifier  =  identifier  .
arguments = [arguments,] ( expression | string-constant
                         | YES | NO  ) .
expression  =  arithmetic-expression
            |  boolean-expression  .
arithmetic-expression  =  ["-"] term
  |  arithmetic-expression "+" term
  |  arithmetic-expression "-" term  .
term  =  factor
      |  term "*" factor
      |  term "/" factor  .
factor  =  primary
        |  primary "**" primary
        |  primary "**-" primary  .
primary  =  integer-constant
         |  real-constant
         |  destination
         |  procedure-call
         |  "(" arithmetic-expression ")"  .

boolean-expression  =  [boolean-expression "|"] boolean-factor  .
boolean-factor  =  [boolean-factor "&"] boolean-secondary .
boolean-secondary  =  ["~"] boolean-secondary
                   |  boolean-primary  .
boolean-primary  =  YES | NO
   |  destination
   |  procedure-call
   |  arithmetic-expression relational-operator arithmetic-expression .
```

If a subscript is empty, it is assumed that an entire row or column is being referenced. If all the subscripts are empty, the parentheses and commas may also be omitted. Array expressions are performed elementwise.

To refer to a procedure without actually invoking it, put a @ before the procedure identifier.

If the operands are mixed INTEGER and REAL the result is REAL; if either is COMPLEX, the result is COMPLEX. Dividing an INTEGER by an INTEGER and raising an INTEGER to a power are illegal.

PRIMITIVES AND MACROS. The following macros are predefined:

```
 LENGTH(array,i)   size of array in the i-th dimension
 DADD, DMUL, DDIV    extended precision arithmetic
 CEIL, FLOOR, SIGN, ABS, FLOAT, RE, IM
 MAX, MIN ( x1, x2, x3, ... )
 MOD ( x1, x2 )     smallest nonnegative r such that
         ( x1 - r ) / x2  is integral
 PI, EPS, MAXREAL, MAXINTEGER
 ACOS, ASIN, ATAN, COS, EXP, LN or LOG,
      LOG10, SIN, SQRT, TAN
```

The following procedures are predefined:

NEXT LINE[(n)]	skip to start of next line, then put out n-1 blank lines (if no argument then n=1)
NEXT PAGE	page eject
END OF INPUT	BOOLEAN expression set to YES when the input file becomes empty
BLANK SEPARATION(n)	number of blanks to be left between output values (default 3)
INTEGER DIGITS(n)	number of digits for integer output (default 12) 1 <= n <= 19
REAL DIGITS(n)	number of significant digits for real output (default 4) 3 <= n <= 15
REAL LEADING DIGITS (n)	0 <= n (default 3)
DATE AND TIME	is replaced by: 'dd mmm 19yy hh:mm'

Macros are evaluated much like procedure calls. First, each argument is evaluated; then the macro is applied to its arguments by inline expansion according to the macro definition; finally, the replacement text is regarded as fresh input. During the evaluation, tokens beginning with "_" have the first "_" stripped off. (This allows macros to be temporarily "hidden.")

DEFINE(identifier, replacement text) defines a macro. The replacement text is a sequence of tokens, possibly containing eor and matched parentheses. Places in the replacement text where arguments are to be inserted during expansion are indicated by $ followed by a digit or letter.
IFELSE(a,b,c,d) is replaced by c if the token a is the same as b, otherwise by d. Either a or b may be empty. For example, the text

```
      DEFINE(VERBOSE,YES)
      IFELSE(VERBOSE,YES,PUT DATA(X),)
```

would be replaced by

```
      PUT DATA(X)
```

A macro defined before the first procedure applies globally; other macros, which may temporarily redefine the global ones, apply only to the procedure in which they are defined.

APPENDIX I

T implementation notes

FILE FORMAT. The Report mentions only an abstract end-of-record delimiter and not any particular file format. This is because the assumptions of local text editors are of overwhelming importance.

The current implementation provides for card image files, in which columns 73 through 80 are ignored. In order to obtain text records longer than 72 characters, continuation lines, flagged by a blank in column 1, may be used. Comments, starting with a "#" end in column 72.

More technically, each card has an end-of-line character EOL appended to it and each record has an EOR appended after the last EOL. Thus

```
  REAL:    # constants
    PI, EPS
```

is translated to

```
 REAL: #constants  EOL  PI, EPS   EOL EOR
```

The T processor discards blanks and text from a # up to the next EOL. The runtime I/O package discards blanks, comments, and begins and ends. On the other hand, the PRINT program makes use of EOLs to intelligently format its listing.

The input/output procedures effectively append an infinite string of EOF characters to the end of the file. The characters EOL, EOR, and EOF are represented internally as the ASCII control characters US, RS, and FS.

OTHER REMARKS ON USING T. Strings may be passed through a procedure, for example from a high level routine to a graphics primitive, by declaring a formal paramter as INTEGER(1).

Various options may be invoked by a macro call of the form:

```
      option( ? )
```

where

? is either ON or OFF

and option is [default value is in brackets]

RECURSIVE	[OFF]	recursive procedures
SUBCHECK	[OFF]	subscript checking
SHORT	[OFF]	short precision
FORTRAN FACADE	[OFF]	procedure looks to the outside world like Fortran
UNDERFLOW	[ON]	turn off underflow error messages in the current procedure and its descendants (reals that underflow are set to 0)

Arrays may have at most 5 subscripts.

For the purposes of STRUCTURE alignment, the sizes of the scalar types are:

INTEGER	1 unit
REAL	2 units
COMPLEX	4 units

REALs and COMPLEXes should start an even number of units into the STRUCTURE for fastest access.

JCL AT SLAC. In order to invoke the PRINT program (designed to intelligently list a set of T procedures), use the PRINT catalogued procedure stored in WYL.CG.EHG.E. An example of a PRINT run:

```
//   JOB
//PROCLIB DD DSN=WYL.CG.EHG.E,DISP=SHR
//P EXEC PRINT
//INPUT DD *
  ... T procedures ...
```

The PRINT program recognizes three commands, which look like comments:

#%E	causes the next line to appear on the next page
#%Ttitle	causes the characters immediately appearing after "#%T" to appear in the title line, and the next line will appear on the next page
#%Ln	causes only identions of at most level n to appear, e.g. #%I0 will cause indention to be suppressed

In order to invoke the precompiler and compile the resulting PL/I, use the TC catalogued procedure stored in WYL.CG.EHG.E. An example of a compile-and-go run:

```
//   JOB
//PROCLIB DD DSN=WYL.CG.EHG.E,DISP=SHR
//C EXEC TC
//INPUT DD *
  ... T procedures ...
//G EXEC PLIG
```

Provision has been made for a PL/I dump. To get this, add the JCL card:

```
//PLIDUMP DD SYSOUT=A,DCB=(RECFM=FBA,LRECL=133,BLKSIZE=1330)
```

APPENDIX G

Report on the graphics interface G

> "In improving exploratory data analysis, we need to find new questions to ask of the data (probably the hardest task), and new ways to ask old questions. Throughout, arithmetic as a basis for preparing pictures is likely to be the keynote. It is most important that we see in the data those things we do not expect -- pictures help us in this far more than numbers, though we can gain a lot just by what numbers we use."
>
> John Tukey

ESTABLISHING THE ENVIRONMENT. In order to remember various options and settings, such as character size and line type, and as a buffer for lowlevel plotting commands, a work area PLOT is provided to the graphics procedures. This may be declared as:

```
STRUCTURE: PLOT
   INTEGER(500): WORK
   STRUCTURE: USER
   ...
```

The workspace USER, which may be as large or small as desired, allows parameters to be passed to user procedures called by G.

To initialize PLOT at the start of a run, call

```
GOPEN( DEVICE, PLOT )
```

where

DEVICE is a string containing one of the codes:

CALFICH	microfiche
PDS4013	Tektronix, Hewlett-Packard
VEP12FF	Versatec Note that if VEP12FF,EXTSORT is specified, then the reordering of the plot commands necessary for the Versatec will not be done by a main memory sort (greatly reducing the run-time memory requirements). In this case an external sort step must be supplied.

Before each new picture, including the first, call

```
GPICT( PLOT )
```

Finally, at the end of the run, call

```
GCLOSE( PLOT )
```

Omitting this may cause the last picture to be lost.

For convenience, the initial transformation is GTRAN1A, which performs the mapping

```
W := SCALE * ( X - X0 )
```

where

```
      REAL(2): W, X, X0, SCALE
```
and X0 = (0,0), SCALE = (1,1). Tc change these values, which are saved in PLOT.WORK, call

```
      GTBAN1( X0, SCALE, PLOT )
```

Line types, character sizes, and character angles are specified by calling

```
      GLTYPE( LTYPE, PLOT )
      GCSIZE( CSIZE, PLOT )
      GCANGL( CANGL, PLOT )
   where
      LTYPE is a string containing one of the codes:
         SOLID, DOT, DASH, or DOT-DASH
      REAL: CSIZE,  # character spacing; initially 1;
            CANGL   # character angle, in radians
                    #  counterclockwise from horizontal;
                    #  initially 0;
```

BASIC DRAWING. To jump straight to the pcint X,

```
      GJUMP( X, PLOT )
   where
      REAL(): X   # destination (in user coordinates)
```

To draw a line from the current position tc X,

```
      GDRAW( X, PLOT )
```

If the ccordinate transformation is curvilinear, this produces a piecewise linear approximation to the implied curve. Following a change of coordinate system, GJUMP should be called before GDRAW.

To write out text, call

```
      GTEXT( X, PRI, SEC, PLOT )
   where
      REAL(): X      # location for center of first character
      PRI, SEC  are strings of length at most 255
```

In order to obtain a large alphabet, text is presented to GTEXT using a pair of strings. Every pair of corresponding characters in the primary and secondary strings denotes one character in the extended alphabet.

The secondary character for

lower Roman	upper Roman	lower Greek	upper Greek
L		G	H

is

For common special characters the secondary character is also a space. The primary character for Greek letters is the first letter of its English name or one of the special cases:

H	eta	F	phi	W	omega
Q	theta	Y	psi		

Additional special and control characters are available:

0C	begin superscript
1	end
2	begin subscript
3	end
4	save position1
5	restore
6	save position2
7	restore
8	save position3
9	restore
E	increase size
F	decrease
1V	half up
2	down
3	third up
4	down
5	sixth up
6	down
0U	one back
1	half forward
2	back
3	third forward
4	back
5	sixth forward
6	back
MX	is an element of
N	is not an element or
E	there exists
A	for all
I	intersect
U	union
<	is strictly contained in
>	strictly contains
L	is contained in
R	contains
UA	up arrow
D	down
L	left
R	right
B	bidirectional

IS	integral
J	contour integral
P	partial-d
D	del
S	plus-or-minus
X	times
:	divided by
+	abstract plus
*	abstract times
2	radical
0	infinity
/	back slash
(	left square bracket
)	right square bracket
L	left angle bracket
R	right angle bracket
A	left curly bracket
Z	right curly bracket
<	less or equal
=	not equal
E	equivalent to
Q	proportional to
>	greater equal
O	degree
N	section
G	dagger
F	double dagger
H	h bar
W	lambda bar
U	underscore
V	overscore
0P	cross
1	diagonal cross
2	diamond
3	box
4	star
5	diagonal start
6	cross with serifs
7	diagonal cross with serifs
8	compass rose
9	octagon

For example, the definition of the gamma function is given by:

```
GTEXT(X,'(N-1)!=G(N)=I40052P0S E0-T1T0N-11DT',
       ' L     H L  SCCSCCC C IC ICICI  CLL',PLOT)
```

DIMENSIONING. Given limits on the data range, suitable values for plotting the data may be obtained by calling

```
GSCALE( DATA MIN, DATA MAX,                          # input
   LABEL MIN, LABEL MAX, EXP, MAJOR, MINOR )  # output
```

The data will then be bracketed by LABEL MIN*10**EXP and LABEL MAX*10**EXP, which are round numbers in engineering format.

GTIC acts somewhat like a ruler, drawing an unlabelled axis with large and small tic marks:

```
GTIC( L, H, MAJOR, MINOR, OFF, PLOT )
```

where

```
REAL(): L, H,  # endpoints of axis;
        OFF    # offset coordinates of major tic mark
               #  endpoints, which determine the size
               #  and direction of the tic marks;
REAL: MAJOR, MINOR  # number of major and minor tic marks,
               #  counting endpoints;  thus a yardstick
               #  might have MAJOR=4, MINOR=13;
```

GLAB adds integer labels to the tic marks produced by GTIC:

```
GLAB( L, H, MAJOR, OFF, LOW, HIGH, PICT )
```

where

```
INTEGER: LOW, HIGH    # label values at endpoints;
REAL(): OFF           # offset of first character of label
```

L, H, MAJOR are as in GTIC.

GFORM1 lays out a form suitable for scatter plots and function graphs:

```
GFORM1( GPRI, GSEC,
        XPRI, XSEC, XLOW, XHIGH,
        YPRI, YSEC, YLOW, YHIGH,  PLOT )
```

where

```
GPRI, GSEC, XPRI, XSEC, YPRI, YSEC are pairs of strings
      defining the general title, X- and Y-axis labels;
REAL: XLOW,XHIGH,YLOW,YHIGH  # specifies the data limits
```

GFORM1 automatically sets up a coordinate system so that the plot will fill the screen. To add a function curve, just use GJUMP and GDRAW.

PLOTTING. GSCAT provides scatter plots:

```
GSCAT( GPRI, GSEC,
       XPRI, XSEC, X,
       YPRI, YSEC, Y,  PLOT )
```

where

```
REAL(): X, Y      # data points
```

GPRI, ... , YSEC are as in GFORM1.

GGRAPH provides graphs of functions of one variable:

```
GGRAPH( GPRI, GSEC,
        XPRI, XSEC, A, B,
        FPRI, FSEC, F,  PLOT )
```

where

```
      REAL: A, B          # endpoints of interval for graphing
      PROCEDURE()REAL: F    # function to be plotted, which
                          # is called in the form:
                          #   Y = F( X, PLOT )
```

To plot contours of the surface passing through the points
(X(I), Y(J), F(I,J))
call

```
      GCONT( LL, UR, F, LEVELS, PLOT )
```

where

```
      REAL(2): LL, UR     #  coordinates of lower left (most
                          #  negative) and upper right (most
                          #  positive) corners of the rectangle
                          #  in the X Y plane on which data
                          #  is given
      REAL(,): F          #  data values
      REAL(): LEVELS      #  contour levels to be plotted
```

A uniform grid is assumed.

To draw a transect surface plot with hidden lines removed, call

```
      GSURF( LL, UR, F,
         AZIMUTH, ELEVATION, ORIGIN, SCALE,  PLOT )
```

where

```
      REAL: AZIMUTH, ELEVATION, SCALE
      REAL(2): ORIGIN
```

The coordinate transformation used in GSURF maps P = (X,Y,F) into

```
   ORIGIN  +  (  -S1      C1       0    )  (P-C)*SCALE
              (  -S2*C1   -S2*S1   C2   )
```

where C1 = COS(AZIMUTH), ..., S2 = SIN(ELEVATION)
C = ((LL(1)+UR(1))/2, (LL(2)+UR(2))/2, 0)

LL, UR, and F are as in GCONT.